Language with Functions & Closures

Environment Version · AST · Value · Env · Parser · Interpreter · Runner

1. Grammar (BNF)

The surface language supports numeric literals, addition, a single-variable local binding form, anonymous functions (**lam**), function application, and variable references. Every form except atoms uses curly-brace lists.

`<expr>`	`::=`	`<num>`	*numeric literal*
	`|`	`{ + <expr> <expr> }`	*addition*
	`|`	`{ lam <var> <expr> }`	*anonymous function*
	`|`	`{ let1 { <var> <expr> } <expr> }`	*local binding*
	`|`	`{ <expr> <expr> }`	*function application*
	`|`	`<var>`	*variable reference*

2. AST

The **Exp** type represents all possible expressions. Constructors are suffixed with **E**. Two new constructors — **lamE** and **appE** — add functions to the language.

```
(define-type Exp
  [numE  (n    : Number)]
  [plusE (left : Exp) (right : Exp)]
  [varE  (name : Symbol)]
  [let1E (var  : Symbol)          ; bound name
         (val  : Exp)             ; value expression
         (body : Exp)]            ; body expression
  [lamE  (var  : Symbol)          ; {lam x body}
         (body : Exp)]
  [appE  (fun  : Exp)             ; {f arg}
         (arg  : Exp)])
```

3. Value Type

The **Value** type has two constructors. A **numV** wraps a number. A **funV** is a *closure*: it stores the parameter name, the body, and — critically — the environment active at the moment of creation. That saved

environment is what gives the language **lexical scope**.

```
(define-type Value
  [numV (n    : Number)]           ; a numeric result
  [funV (var  : Symbol)            ; closure: parameter name
        (body : Exp)               ;          function body
        (nv   : Env)])             ;          saved environment  <-- key!
```

4. Environment

An **Env** is an immutable list of **Binding** pairs. **extend** conses a new binding onto the front — never mutates. Because newer bindings are at the front, **lookup** naturally implements shadowing by returning the first match.

```
(define-type Binding
  [bind (name : Symbol)
        (val  : Value)])

(define-type-alias Env (Listof Binding))

(define empty-env : Env empty)     ; the empty environment
```

5. lookup (Symbol Env → Value)

Walks the list front-to-back. Returns the first matching value; raises an error if the name is unbound. The front-to-back walk means inner bindings always shadow outer ones.

```
;; lookup : Symbol Env -> Value
(define (lookup [name : Symbol] [env : Env]) : Value
  (cond
    [(empty? env)
     (error name "unbound variable")]
    [(symbol=? name (bind-name (first env)))
     (bind-val (first env))]
    [else
     (lookup name (rest env))]))
```

6. extend (Env Symbol Value → Env)

Returns a new environment with the new binding at the front. The original environment is never modified.

```
;; extend : Env Symbol Value -> Env
(define (extend [env  : Env]
                [name : Symbol]
                [val  : Value]) : Env
  (cons (bind name val) env))
```

7. Parser (S-Exp → Exp)

Converts surface syntax into an AST. The parser is purely structural — it recognises syntax and builds constructors with no knowledge of evaluation. Any list whose head is not a known keyword is treated as an **appE** (function application).

```
(define (parse [s : S-Exp]) : Exp
  (cond
    ; atom: number
    [(s-exp-number? s)  (numE (s-exp->number s))]

    ; atom: variable name
    [(s-exp-symbol? s)  (varE (s-exp->symbol s))]

    ; list: inspect the head
    [(s-exp-list? s)
     (let ([l (s-exp->list s)])
       (cond
         ; {+ l r}
         [(symbol=? '+ (s-exp->symbol (first l)))
          (plusE (parse (second l)) (parse (third l)))]

         ; {lam x body}
         [(symbol=? 'lam (s-exp->symbol (first l)))
          (lamE (s-exp->symbol (second l))
                (parse (third l)))]

         ; {let1 {x val} body}
         [(symbol=? 'let1 (s-exp->symbol (first l)))
          (let ([binding (s-exp->list (second l))])
            (let1E (s-exp->symbol (first binding))
                   (parse (second binding))
                   (parse (third l))))]

         ; {f arg}  -- anything else is a function call
         [else
          (appE (parse (first l))
                (parse (second l)))]))]

    [else (error 'parse "not a number, symbol, or list")]))
```

8. Interpreter (Exp Env → Value)

Pattern-matches on the AST and returns a **Value**. The two new cases are the heart of the chapter:

lamE — creates a closure by packaging the parameter, body, and the *current* environment **nv** into a **funV**. The environment is frozen at definition time.

appE — evaluates function and argument, then evaluates the body in the closure's *saved* environment extended with the argument. The pattern **(funV v b nv*)** extracts and shadows the outer **nv**, so the closure's environment is used rather than the caller's — this is the single act that enforces **lexical scope**.

```
(define (interp [e : Exp] [nv : Env]) : Value
  (type-case Exp e

    [(numE n)   (numV n)]

    [(varE s)   (lookup s nv)]

    [(plusE l r) (add (interp l nv) (interp r nv))]

    ; lam: create a closure — nv is saved inside funV!
    [(lamE v b)  (funV v b nv)]

    ; app: evaluate fun & arg, then use the CLOSURE's saved env
    [(appE f a)
     (let ([fv (interp f nv)]
           [av (interp a nv)])
       (type-case Value fv
         [(funV v b nv*)            ; nv* shadows outer nv!
          (interp b (extend nv* v av))]
         [else (error 'app "not a function")]))]

    [(let1E var val body)
     (let ([new-env (extend nv var (interp val nv))])
       (interp body new-env))]))
```

9. Helper: add (Value Value → Value)

Unwraps two **numV** values, adds them, and re-wraps the result. Raises an error if either argument is not a number.

```
;; add : Value Value -> Value
(define (add [l : Value] [r : Value]) : Value
  (cond
    [(and (numV? l) (numV? r))
     (numV (+ (numV-n l) (numV-n r)))]
    [else (error 'add "not a number")]))
```

10. Runner (S-Exp → Value)

Composes **parse** and **interp** into a single entry point. Every program starts from the empty environment.

```
(define (run [s : S-Exp]) : Value
  (interp (parse s) empty-env))    ; always start with empty env
```

11. Tests

```
(print-only-errors #true)

; --- numbers and addition ---
(test (run '3)                      (numV 3))
(test (run '{+ 3 4})                (numV 7))
(test (run '{+ {+ 1 2} {+ 3 4}})    (numV 10))

; --- variables and let1 ---
(test (run '{let1 {x 3} x})         (numV 3))
(test (run '{let1 {x 3} {+ x x}})   (numV 6))

; --- nested let1: inner x shadows outer x ---
(test (run '{let1 {x 1}
                  {let1 {x 2} x}})
      (numV 2))

; --- lambda: immediate application ---
(test (run '{{lam x {+ x x}} 3})    (numV 6))

; --- bind lambda to name, then call ---
(test (run '{let1 {sq {lam x {+ x x}}}
                  {sq 5}})
      (numV 10))

; --- lexical scope: f captures x=3, later x=2 is ignored ---
(test (run '{let1 {x 3}
                  {let1 {f {lam y {+ x y}}}
                        {let1 {x 2}
                              {f 4}}}})
      (numV 7))    ; 3+4=7, NOT 2+4=6

; --- closure returned and used outside its defining scope ---
(test (run '{{let1 {x 3} {lam y {+ x y}}} 4})
      (numV 7))

; --- errors ---
(test/exn (run 'x)                  "unbound")
(test/exn (run '{f 3})              "unbound")
(test/exn (run '{{lam x x} {lam x x}})  "not a number")
```

12. Key Concepts

Concept	Code	What it means

Closure creation	`(funV v b nv)`	Saves the current environment inside the function value
Closure's env	`(funV v b nv*)`	Extracts saved env; nv* shadows the outer nv
Lexical scope	`extend nv* v av`	Body runs in definition-time env, not caller's env
Dynamic scope ✗	`extend nv v av`	Would use caller's env — the bug this design avoids
Shadowing	`cons (bind n v) env`	Newer bindings at front; lookup returns first match

13. Pipeline Trace

Step-by-step execution of **{let1 {f {lam x {+ x x}}} {f 3}}** through the full **run** → **parse** → **interp** pipeline.

Step 1 · run is called

```
(run '{let1 {f {lam x {+ x x}}} {f 3}})
=> (interp (parse '{let1 {f {lam x {+ x x}}} {f 3}}) empty-env)
```

Step 2 · parse builds the AST

```
Input: {let1 {f {lam x {+ x x}}} {f 3}}
  head = 'let1  =>  let1E branch
    binding  = {f {lam x {+ x x}}}
    var-name = 'f
    var-val  = parse({lam x {+ x x}})
                 => lamE 'x (parse {+ x x})
                 => lamE 'x (plusE (varE 'x) (varE 'x))
    body     = parse({f 3})
                 => else branch  =>  appE
                 => appE (varE 'f) (numE 3)

parse output:
  (let1E 'f
         (lamE 'x (plusE (varE 'x) (varE 'x)))
         (appE (varE 'f) (numE 3)))
```

Step 3 · interp receives let1E and empty-env

```
(interp (let1E 'f (lamE ...) (appE ...)) empty-env)
let1E branch fires:

  step 3a: evaluate val-expr in current env
    (interp (lamE 'x (plusE (varE 'x) (varE 'x))) empty-env)
    lamE branch => (funV 'x body empty-env)   <-- closure created!

  step 3b: extend the environment
    (extend empty-env 'f (funV 'x body empty-env))
    new-env = { f -> (funV 'x body empty-env) }

  step 3c: evaluate body in new-env
    (interp (appE (varE 'f) (numE 3)) new-env)
```

Step 4 · appE fires

```
fv = (interp (varE 'f) new-env)
     = (lookup 'f new-env)
     = (funV 'x (plusE (varE 'x) (varE 'x)) empty-env)

av = (interp (numE 3) new-env) = (numV 3)

pattern match: (funV v b nv*)
  v   = 'x
  b   = (plusE (varE 'x) (varE 'x))
  nv* = empty-env     <-- closure's saved env (shadows outer nv!)

(interp b (extend nv* 'x (numV 3)))
= (interp (plusE (varE 'x) (varE 'x)) { x -> (numV 3) })
```

Step 5 · evaluate the body

```
plusE branch:
  LEFT:  (interp (varE 'x) { x->(numV 3) }) = (numV 3)
  RIGHT: (interp (varE 'x) { x->(numV 3) }) = (numV 3)
  add:   (numV (+ 3 3)) = (numV 6)
```

Final Output

```
(numV 6)
```